Memoria

Práctica 1

Desarrollo de Software

3º Ingeniería Informática - 2023/2024

Índice

1. Patrón Factoría Abstracta.

1.1 Planteamiento del problema

Por un lado, para separar la creación de objetos de su uso, se han creado las clases abstractas *FactoriaMontaña* y *FactoriaCarretera* que heredan de la interfaz *FactoriaCarreraYBicicleta*, de esta forma se pueden crear objetos sin tener que especificar su clase.

Por otro lado, usando el patrón creacional Prototipo, los objetos se crean por clonación de la instancia que se quiera, o bien a partir de la clase *CarreraMontaña* o *CarreraCarretera* que heredan de la clase abstracta *Carrera*, o bien a partir de *BicicletaMontaña* o *BicicletaCarretera* que heredan de la clase abstracta *Bicicleta*.

1.2 Diagrama de diseño

Por un lado, para separar la creación de objetos de su uso, se han creado las clases abstractas *FactoriaMontaña* y *FactoriaCarretera* que heredan de la interfaz *FactoriaCarreraYBicicleta*, de esta forma se pueden crear objetos sin tener que especificar su clase.

Por otro lado, usando el patrón creacional Prototipo, los objetos se crean por clonación de la instancia que se quiera, o bien a partir de la clase *CarreraMontaña* o *CarreraCarretera* que heredan de la clase abstracta *Carrera*, o bien a partir de *BicicletaMontaña* o *BicicletaCarretera* que heredan de la clase abstracta *Bicicleta*.

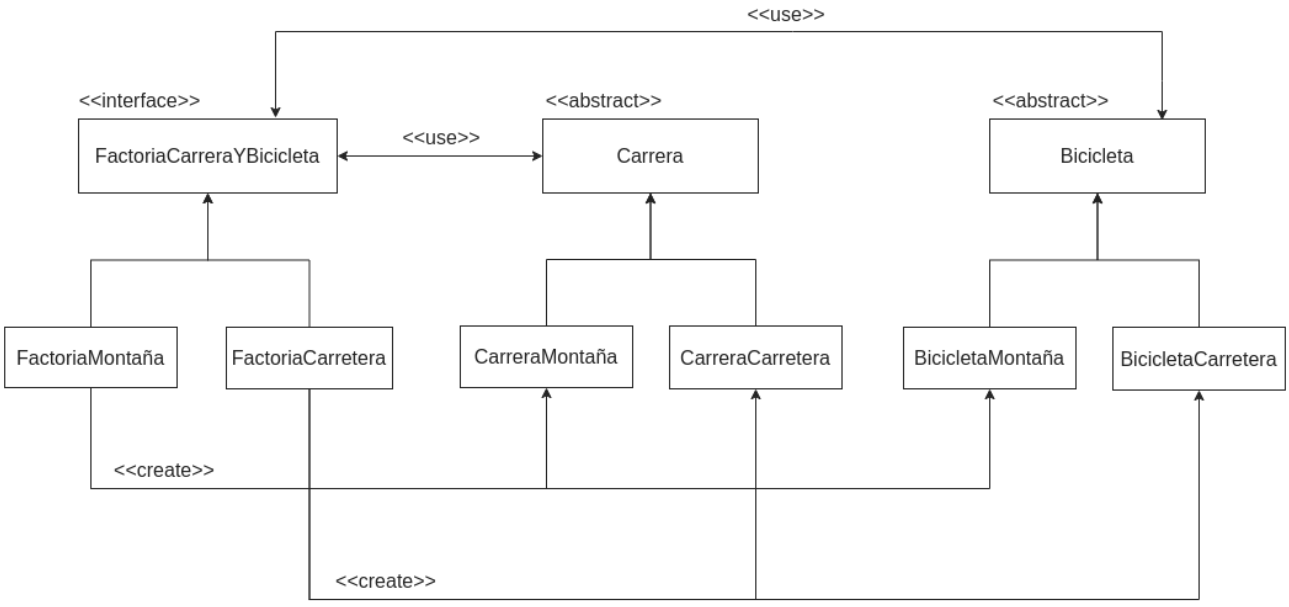

Ilustración X. Diagrama de diseño, ejercicio 2.

1.3 Solución del problema

1.4 Conflictos encontrados

2. Patrón Factoría Abstracta + Patrón Prototipo.

2.1 Planteamiento del problema

Diseñar e implementar una aplicación con la misma funcionalidad que el ejercicio anterior, aplicando el patrón Factoría Abstracta junto con el patrón Prototipo. Para desarrollar esta aplicación se ha elegido el lenguaje de programación **Python** y se ha llevado a cabo con el uso de hebras.

2.2 Diagrama de diseño

Por un lado, para separar la creación de objetos de su uso, se han creado las clases abstractas *FactoriaMontaña* y *FactoriaCarretera* que heredan de la interfaz *FactoriaCarreraYBicicleta*, de esta forma se pueden crear objetos sin tener que especificar su clase.

Por otro lado, usando el patrón creacional Prototipo, los objetos se crean por clonación de la instancia que se quiera, o bien a partir de la clase *CarreraMontaña* o *CarreraCarretera* que heredan de la clase abstracta *Carrera*, o bien a partir de *BicicletaMontaña* o *BicicletaCarretera* que heredan de la clase abstracta *Bicicleta*.

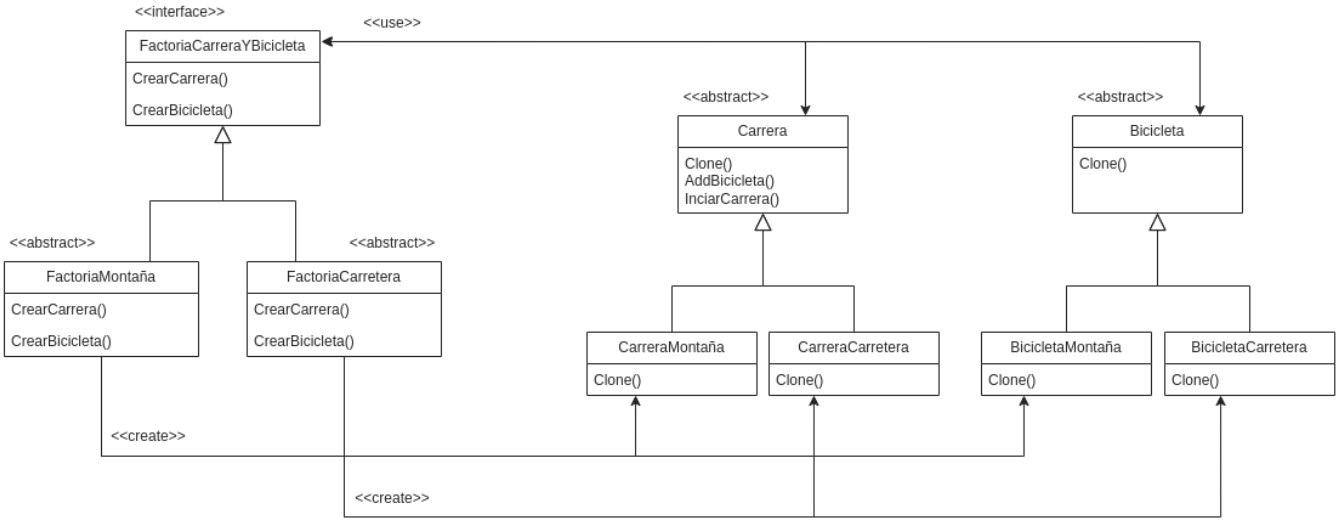

Ilustración X. Diagrama de diseño, ejercicio 2.

2.3 Solución del problema

Este ejercicio se ha intentado resolver tanto sin hebras como con hebras, primero se va a explicar la jerarquía de clases puesto que es la misma en ambos casos y posteriormente los programas principales que la utilizan. Se evitará incluir las importaciones de módulos.

La interfaz **FactoriaCarreraBicicleta** es muy simple, al igual que en el ejercicio anterior tiene dos métodos abstractos CrearCarrera y CrearBicicleta que serán las subclases las que los implementen.

```
class FactoriaCarreraBicicleta(ABC):

    @abstractmethod
    def crear_carrera(self):
        pass

    @abstractmethod
    def crear_bicicleta(self):
        pass
```

Las subclases **FactoriaCarretera** y **FactoriaMontaña** son muy parecidas, pero permiten crear familias de objetos relacionados distintos entre sí. FactoriaCarretera permitirá crear objetos CarreraCarretera o BicicletaCarretera.

```
class FactoriaCarretera(FactoriaCarreraBicicleta):

    def crear_carrera(self, bicicletas, porcentaje):
        return CarreraCarretera(bicicletas, porcentaje)

    def crear_bicicleta(self, id):
        return BicicletaCarretera(id)
```

La clase **CarreraCarretera** que hereda de **Carrera**, en su constructor llama a la clase padre para inicializar el objeto. A diferencia del ejercicio anterior, se implementa un método para realizar una copia profunda del objeto CarreraCarretera que devuelve un objeto idéntico al original. La clase **CarreraMontaña** es muy parecida, diferenciandose unicamente en el tipo de objeto.

```
class CarreraCarretera(Carrera):

    def __init__(self, bicicletas, porcentaje):
        super().__init__(bicicletas, porcentaje)

    def __deepcopy__(self, memo):
        new = CarreraCarretera(deepcopy(self.bicicletas, memo),
                               deepcopy(self.porcentaje, memo))
        return new
```

La clase **BicicletaCarretera** que hereda de **Bicicleta**, en su constructor llama a la clase padre para inicializar el objeto. A diferencia del ejercicio anterior, se implementa un método para realizar una copia profunda del objeto BicicletaCarretera que devuelve un objeto idéntico al original. La clase **BicicletaMontaña** es muy parecida, diferenciandose unicamente en el tipo del objeto.

```
class BicicletaCarretera(Bicicleta):

    def __init__(self, id):
        super().__init__(id)

    def __deepcopy__(self, memo):
        new = BicicletaCarretera(deepcopy(self.id, memo))
        return new
```

La clase abstracta **Carrera** finalmente construye los objetos CarreraCarretera y CarreraMontaña con sus correspondientes atributos. Sin embargo, el método de copia se tiene que implementar en las subclases puesto que no es lo mismo un objeto de CarreraCarretera que un objeto CarreraMontaña.

```
class Carrera(ABC):

    def __init__(self, bicicletas, porcentaje):
        self.bicicletas = bicicletas
        self.porcentaje = porcentaje

    @abstractmethod
    def __deepcopy__(self, memo):
        pass

    def add_bicicleta(self, bicicleta):
        self.bicicletas.append(bicicleta)

    def iniciar_carrera(self, name):
        # Dormir la hebra
        sleep(60)
        # Retirar las bicis
        retirar_n_bicis = math.ceil(self.porcentaje*num_bicis)
        ...
```

La clase abstracta **Bicicleta** es muy simple también, construye el objeto BicicletaCarretera o BicicletaMontaña y el método de copia profunda se implementa en las subclases.

```
class Bicicleta(ABC):

    def __init__(self, id):
        self.id = id

    @abstractmethod
    def __deepcopy__(self, memo):
        pass
```

Una vez comentada la jerarquía de clases se van a detallar los resultados obtenidos al ejecutar los distintos programas.

En el programa **main.py** se crean 2 factorias distintas, con el módulo random se determina el tamaño del problema (N) delimitado entre 10 y 40 y, por último, se construyen N bicicletas de cada tipo, N con la factoriaMontaña y N con la factoriaCarretera. Tras crear las carreras, se inician y como se puede ver sólo un porcentaje concreto de bicicletas terminan la carreras.

```
-----------------------------------------------------------------------------------------------
El numero de bicis en las carreras son:  27
-----------------------------------------------------------------------------------------------
Carrera en montaña                          [Thread_ID] = 88012 [Thread_name] = MainThread
Carrera finalizada por 21 bicis             [Thread_ID] = 88012 [Thread_name] = MainThread
Carrera en carretera                        [Thread_ID] = 88012 [Thread_name] = MainThread
Carrera finalizada por 24 bicis             [Thread_ID] = 88012 [Thread_name] = MainThread
-----------------------------------------------------------------------------------------------
```

Ilustración X. Salida programa main.py.

Mientras que el el programa **main_thread.py** se crean dos hebras que ejecutan cada una el método IniciarCarrera, una inicia una carrera de montaña y la otra una carrera de carretera. En el siguiente apartado se explicaran las dificultades encontradas a la hora de utilizar hebras en Python.

```
-----------------------------------------------------------------------------------------------
El numero de bicis en las carreras son:  39
-----------------------------------------------------------------------------------------------
Carrera en montaña                          [Thread_ID] = 91540 [Thread_name] = Thread-1
Carrera en carretera                        [Thread_ID] = 91541 [Thread_name] = Thread-2
Carrera finalizada por 35 bicis             [Thread_ID] = 91541 [Thread_name] = Thread-2
Carrera finalizada por 31 bicis             [Thread_ID] = 91540 [Thread_name] = Thread-1
-----------------------------------------------------------------------------------------------
```

Ilustración X. Salida programa main_thread.py.

2.4 Conflictos encontrados

Para poder trabajar con hebras se ha utilizado la clase *threading*, pero ha sido necesario crear dos subclases de tipo *threading.Thread* para modificar su método run y que así las carreras sean iniciadas por distintas hebras.

En un principio, se intentó ejecutar la función InciarCarrera al crear las hebras con la opción target=carrera_mont.iniciar_carrera('en montaña'), pero de esta forma es la hebra principal (MainThread) la que ejecuta ambas llamadas.

Investigando un poco más encontramos esta manera de solventar el problema.

```
class Thread1(threading.Thread):
    def run(self):
        carrera_mont.iniciar_carrera('en montaña  ')

class Thread2(threading.Thread):
    def run(self):
        carrera_carr.iniciar_carrera('en carretera')

if __name__ == "__main__":

    bicicletas_mont = []
    bicicletas_carr = []

    factoria_mont = fm.FactoriaMontana()
    factoria_carr = fc.FactoriaCarretera()

    n_bicis = random.randrange(10, 40, 1)

    for bici in range(n_bicis):
        bm = factoria_mont.crear_bicicleta(bici+1)
        bicicletas_mont.append(bm)
        bc = factoria_carr.crear_bicicleta(bici+1)
        bicicletas_carr.append(bc)

    print("-------------------------------------------------------------")
    print("El numero de bicis en las carreras son: ", n_bicis)
    print("-------------------------------------------------------------")

    carrera_mont = factoria_mont.crear_carrera(bicicletas_mont, 0.2)
    carrera_carr = factoria_carr.crear_carrera(bicicletas_carr, 0.1)

    # Creamos las hebras
    thread1 = Thread1()
    thread2 = Thread2()

    thread1.start()
    thread2.start()
```

```
thread1.join()
thread2.join()

print("------------------------------------------------------------")
```

3. Patrón libre: patrón Builder.

3.1 Planteamiento del problema

En este ejercicio hemos decidido utilizar el **patrón Builder**, en concreto, tenemos el *PizzaBuilder* y el *BocataBuilder*. De esta forma el director puede elegir si hacer una pizza o un bocata, y el constructor correspondiente crea el tipo de alimento que se quiere. Así se consigue separar la representación del objeto *Pizza* o *Bocata*, de su construcción llevada a cabo por los builders.

3.2 Diagrama de diseño

El constructor *PizzaBuilder* se encarga de crear tres tipos distintos de pizzas: *PizzaInfantilBuilder*, *VeggiePizzaBuilder* y *BBQPizzaBuilder*. Estas pizzas son representadas por la clase *Pizza* cuyos atributos son los ingredientes, salsa, tipo de masa y tamaño de la pizza.

El constructor *BocataBuilder* es muy parecido pero permite crear distintos tipos de bocatas y son representados por la clase *Bocata*.

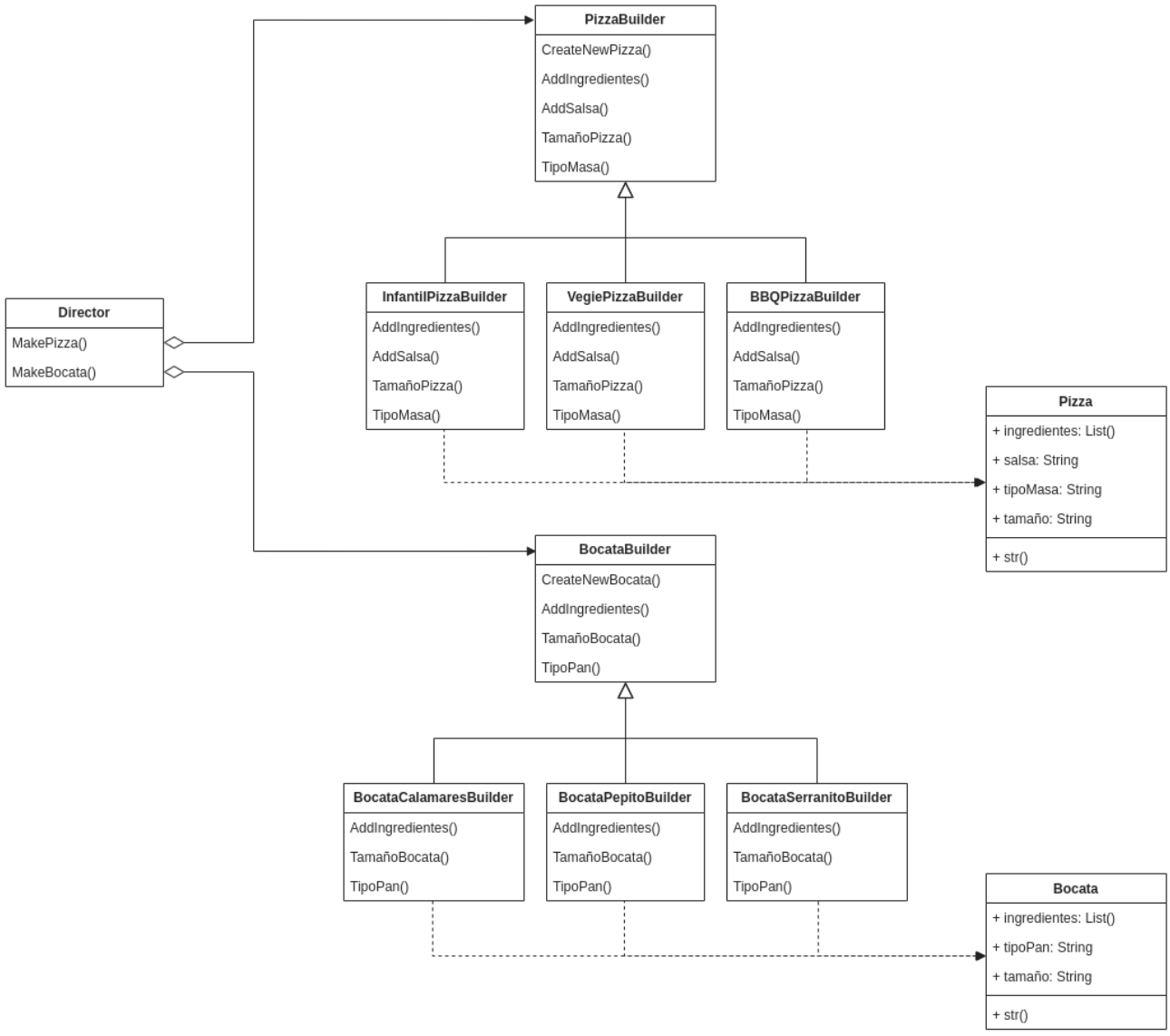

Ilustración X. Diagrama de diseño, ejercicio 3.

3.3 Solución del problema

Para simplificar se va a detallar la estructura de la familia de objetos bocatas, empezando por la clase **BocataBuilder** que es una clase abstracta encargada de constriur el objeto *Bocata* y tiene tres métodos abstractos que implementarán sus subclases. Se evitará incluir las importaciones de módulos para no extender la memoria.

```
class BocataBuilder(ABC):

    def __init__(self):
        self.bocata = None

    def create_new_bocata(self):
        self.bocata = Bocata()

    @abstractmethod
    def add_ingredientes(self):
        pass

    @abstractmethod
    def tipo_pan(self):
        pass

    @abstractmethod
    def tamano_bocata(self):
        pass
```

Luego tenemos las subclases **BocataSerranitoBuilder**, **BocataCalamaresBuilder** y **BocataPepitoBuilder** que heredan de la clase **BocataBuilder** y que se encargan de iniciarlizar los atributos especificos del objeto bocata. En el siguiente fragmento de código se puede visualizar dicha inicialización por parte del *BocataSerranitoBuilder*.

```
class BocataSerranitoBuilder(BocataBuilder):

    def add_ingredientes(self):
        self.bocata.ingredientes = {'aceite', 'tomate', 'jamon serrano',
                                    'lomo', 'pimiento'}

    def tipo_pan(self):
        self.bocata.pan = 'casero'

    def tamano_bocata(self):
        self.bocata.tamano = 'mediano'
```

La clase **Bocata** representa el objeto complejo final y tiene un método *str* para describir las propiedades del objeto, más adelante se expondrán ejemplos de la salida.

```
class Bocata:
    def __init__(self):
        self.ingredientes = {}
        self.pan = None
        self.tamano = None

    def __str__ (self):
        return f"Bocata\n Ingredientes: {self.ingredientes}\n
                 Pan: {self.pan}\n Tamaño: {self.tamano}"
```

La clase **Director** es quien decide que se construya la *Pizza* o el *Bocata* y llama al builder concreto que se le haya pasado como parámetro en su constructor.

```
class Director:
    def __init__(self, builder):
        self._builder = builder

    def build_food(self, food):
        if (food == 'Pizza'):
            self._builder.create_new_pizza()
            self._builder.add_ingredientes()
            self._builder.add_salsa()
            self._builder.tamano_pizza()
            self._builder.tipo_masa()
        else:
            self._builder.create_new_bocata()
            self._builder.add_ingredientes()
            self._builder.tipo_pan()
            self._builder.tamano_bocata()
```

Por último, en el programa principal **main.py** se inicializa el constructor concreto, por ejemplo, el *BBQPizzaBuilder*, luego se crea el Director pasandole el builder, se construye la pizza y, finalmente, se imprime la descripción del objeto.

```
if __name__ == "__main__":

    print("----------------------------------------------------------------------")
    print("Pidiendo una pizza de barbacoa y un serranito...")
    print("----------------------------------------------------------------------")

    BBQBuilder = BBQPizzaBuilder()
    pedido3 = Director(BBQBuilder)
    pedido3.build_food('Pizza')
    print(BBQBuilder.pizza)

    print("----------------------------------------------------------------------")

    BocataSerranito = BocataSerranitoBuilder()
    pedido6 = Director(BocataSerranito)
    pedido6.build_food('Bocata')
    print(BocataSerranito.bocata)

    print("----------------------------------------------------------------------")
```

La salida del programa principal es la siguiente:

```
----------------------------------------------------------------------------------------------------
Pidiendo una pizza de barbacoa y un serranito...
----------------------------------------------------------------------------------------------------
Pizza
 Ingredientes: {'bacon', 'queso', 'tomate', 'jamon york', 'cebolla'}
 Salsa: salsa barbacoa
 Masa: normal con bordes rellenos de queso
 Tamaño: mediana
----------------------------------------------------------------------------------------------------
Bocata
 Ingredientes: {'pimiento', 'aceite', 'jamon serrano', 'tomate'}
 Pan: casero
 Tamaño: mediano
----------------------------------------------------------------------------------------------------
```

Ilustración X. Salida programa main.py.

3.4 Conflictos encontrados

En esta ocasión no hemos encontrado dificultades a la hora de resolver el ejercicio propuesto. Este patrón es un patrón creacional fácil de comprender e implementar.

4. Patrón arquitectónico: Filtros de Intercepción.

4.1 Planteamiento del problema

Diseñar e implementar una aplicación que simule el salpicadero de un coche, respondiendo con la velocidad angular, lineal y kilometraje a actos como encenderlo/apagarlo, acelerar y frenar. Para esto se requiere usar el patrón **Filtros de Intercepción** para definir la modificación de las RPM producidas por la aceleración o el freno y el rozamiento.

Para este ejercicio, por falta de tiempo y de experiencia con Python y Tkinter, nos hemos decantado por implementar el patrón en **Java** siguiendo las indicaciones del guion.

4.2 Diagrama de diseño

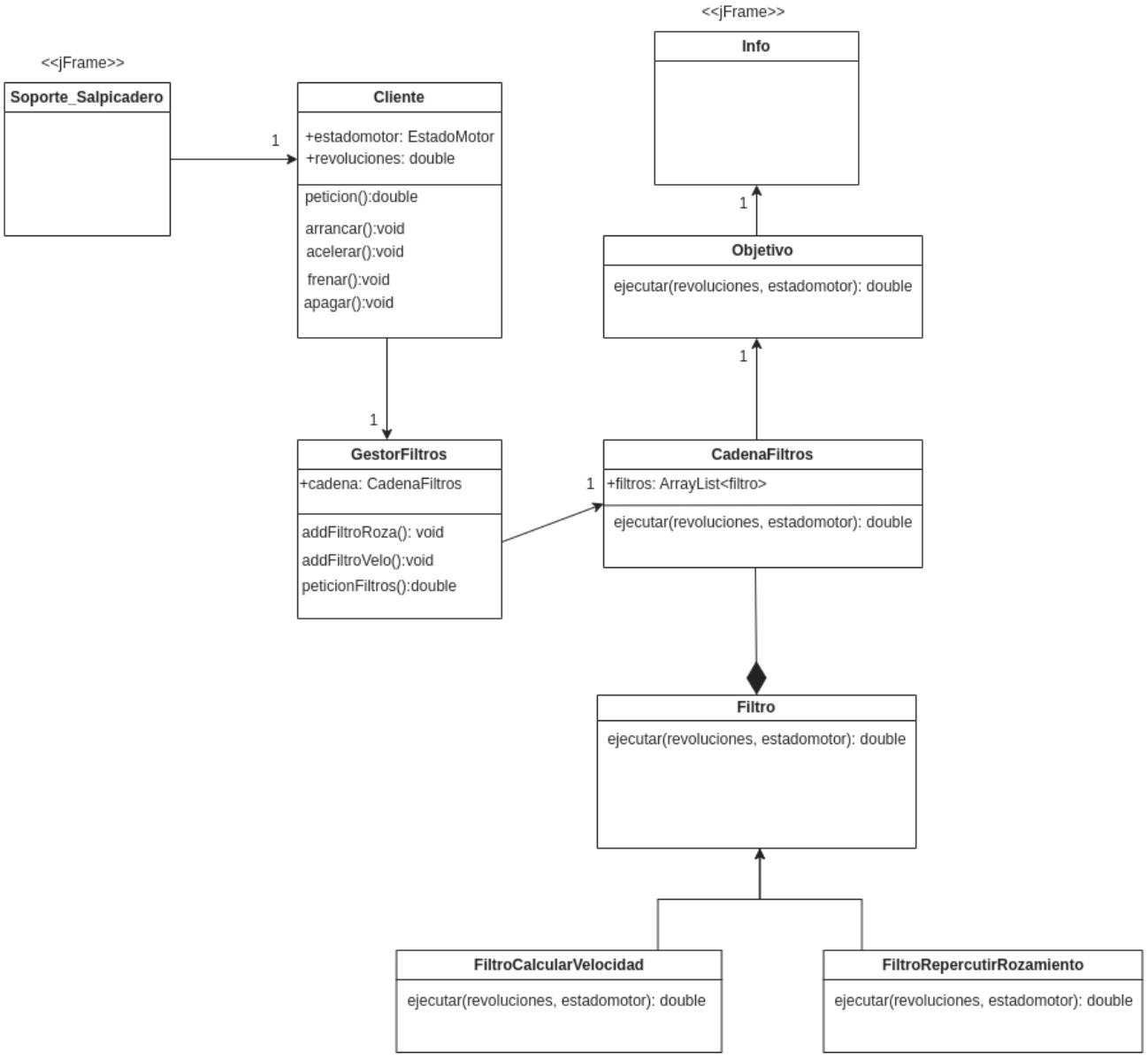

Ilustración X. Diagrama de diseño, ejercicio 4.

4.3 Solución del problema

Como se puede ver en el diagrama de arriba, en nuestra solución la clase **Soporte_Salpicadero** es a partir de la cual ocurre todo. Es del tipo *jFrame*, por lo que además de crear e inicializar al cliente que ordenará al resto de elementos, también mostrará una interfaz gráfica con la que podremos modificar el estado de nuestro coche arrancándolo, acelerando, frenando y apagándolo.

Volviendo al cliente, en el constructor del salpicadero crearemos uno y le ordenaremos que, a su vez, ordene a su *gestor de filtros* que cree dos de ellos para meterlos en la cadena, uno para la aceleración y el frenado y otro para calcular las revoluciones perdidas por el rozamiento.

```
public class Soporte_Salpicadero extends javax.swing.JFrame {

    Cliente cliente;

    public Soporte_Salpicadero() {
        cliente = new Cliente();
        cliente.gestor.addFiltroRoza();
        cliente.gestor.addFiltroVelo();
        initComponents();
    }

    // Código generado automaticamente por NetBeans
    @SuppressWarnings("unchecked")
    private void initComponents() {

        // Creación de los componentes de la ventana Salpicadero
        jLabel1 = new javax.swing.JLabel();
        jToggleButton1 = new javax.swing.JToggleButton();
        ...

        // Configuración y especificación de los componentes de la ventana
        jLabel1.setForeground(new java.awt.Color(255, 0, 0));
        jLabel1.setText("APAGADO");
        ...

        // Los manejadores de eventos, por ejemplo, para encender el motor
        jToggleButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jToggleButton1ActionPerformed(evt);
            }
        });
        ...
    }

    // Método llamado cada vez que se presiona el botón encender
    private void jToggleButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        if("ENCENDER".equals(jToggleButton1.getText())){
            jToggleButton1.setText("APAGAR");
            jLabel1.setText("ENCENDIDO");
            cliente.arrancar();
            cliente.peticion();
        }
        else{
            jToggleButton1.setText("ENCENDER");
            jLabel1.setText("APAGADO");
            cliente.apagar();
            cliente.peticion();
        }
    }
    ...

    // Declaración de variabels
    private javax.swing.JLabel jLabel1;
    private javax.swing.JToggleButton jToggleButton1;
    ...
}
```

La clase **Cliente**, por su parte, es la que modifica el estado del enumerado *EstadoMotor*, es decir, el estado del coche. Además, esta es la que tiene la información de las revoluciones que tiene nuestro coche en cada momento y, cuando queremos mostrarlas por pantalla, es quien se encarga de pasar esa información al gestor para que él haga lo mismo y al final se le devuelva cualquier modificación.

```
public class Cliente {

    EstadoMotor estadomotor;
    double rev;
    GestorFiltros gestor;

    public Cliente(){
        gestor = new GestorFiltros();
        estadomotor = EstadoMotor.APAGADO;
        rev = 0;
    }
```

```
    // Los metodos acelerar, frenar y apagar son muy parecidos a arrancar,
    // lo único que cambia es el estado del motor.
    public void arrancar(){
        estadomotor = EstadoMotor.ENCENDIDO;
    }
    ...

    public void peticion(){
        rev = gestor.peticionFiltros(rev, estadomotor);
    }
}
```

En cuanto al objeto de la clase **GestorFiltros**, es la que se ocupa de añadir los filtros que le ha ordenado el cliente al objeto *CadenaFiltros* y, cuando el mismo cliente inicia la petición de valores, quien pasa a la cadena tanto el estado del motor como las revoluciones.

```
public class GestorFiltros {

    CadenaFiltros cadena;

    public GestorFiltros(){
        cadena = new CadenaFiltros();
    }

    public void addFiltroRoza(){
        FiltroRepercutirRozamiento fil = new FiltroRepercutirRozamiento();
        cadena.filtros.add(fil);
    }

    public void addFiltroVelo(){
        FiltroCalcularVelocidad fil = new FiltroCalcularVelocidad();
        cadena.filtros.add(fil);
    }

    public double peticionFiltros(double revoluciones, EstadoMotor estadomotor){
        return cadena.ejecutar(revoluciones, estadomotor);
    }
}
```

CadenaFiltros contiene un array con todos los filtros que el gestor le haya creado (en este caso 2) y un objeto de la clase *Objetivo*. Esta clase solo tiene un método, pero es el que más trabajo conlleva: primero recorre el array de filtros y pasa a cada uno la información que el gestor le ha dado previamente, recogiendo cada modificación que los filtros le hagan a las revoluciones. Tras esto, estas revoluciones modificadas se las pasa al objetivo para que este las muestre por pantalla junto con la velocidad lineal y los kilómetros recorridos.

```
public class CadenaFiltros {

    public ArrayList<Filtro> filtros;
    public Objetivo obj;

    public CadenaFiltros(){
        obj = new Objetivo();
        filtros = new ArrayList<Filtro>();
    }

    public double ejecutar(double revoluciones, EstadoMotor estadomotor){
        for (int i = 0; i < filtros.size(); i++){
            revoluciones +=filtros.get(i).ejecutar(revoluciones, estadomotor);
        }

        return obj.ejecutar(revoluciones, estadomotor);
    }
}
```

Hablando de este último objeto, **Objetivo**, su única peculiaridad es que a su vez contiene otro jFrame llamado *Info*, y su método “ejecutar” produce que se muestre por pantalla esta ventana, que es la que enseñará los datos calculados también en el objeto de la clase Info.

```
public class Objetivo {

    public double ejecutar(double revoluciones, EstadoMotor estadomotor){

        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Info(revoluciones).setVisible(true);
            }
        });
        return revoluciones;
    }
}
```

Para finalizar, todo este ejercicio tenía la idea de utilizar el patrón de Filtros de Intercepción, y estos se implementan precisamente en la clase interface **Filtro** y sus dos herederas, *FiltroCalcularVelocidad* y *FiltroRepercutirRozamiento*.

Filtro tiene un solo método abstracto, ejecutar, que en FiltroRepercutirRozamiento se basa en restarle 2.5 a las revoluciones dadas y en FiltroCalcularVelocidad, sumarle o restarle 100 siempre que este acelerando o frenando, respectivamente.

Creemos que conseguimos implementar con éxito el patrón ya que nuestro Cliente, cuando quiere información, llama al Gestor y este crea tanto la Cadena como los Filtros, dejando que esta primera sea la que maneje los segundos y tras esto lleve la información modificada al Objetivo.

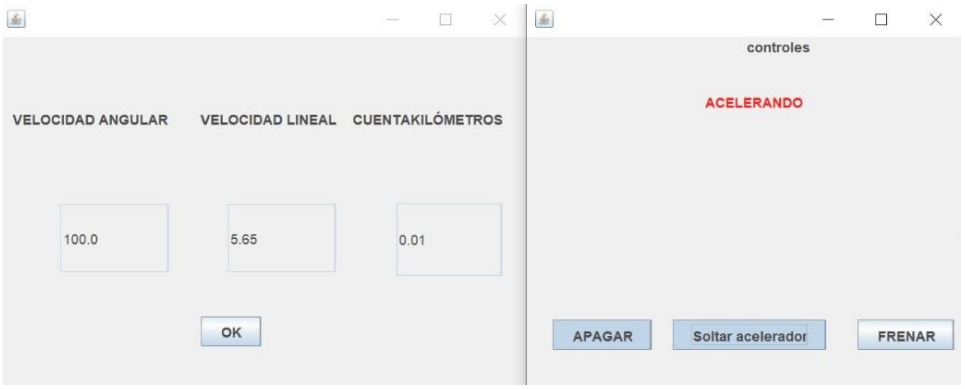

Ilustración X. Salida del programa main.java

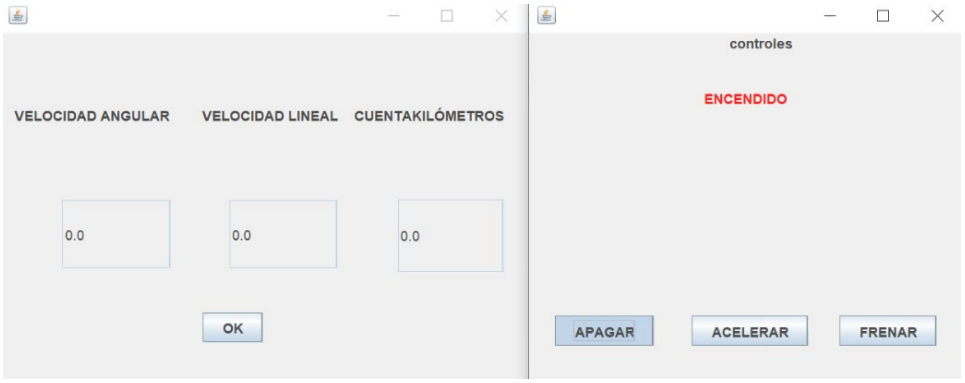

Ilustración X. Salida del programa main.java

Como vemos en las ilustraciones hay un botón que se encarga de encender y apagar el motor, otro para acelerar y dear de hacerlo y un tercero hacer lo mismo con el freno. Cada vez que se pulsa uno de estos tres botones, salvo en el caso de Soltar acelerador y Soltar freno, aparece la segunda ventana donde se muestra en los recuadros el valor que aparece arriba, las RPM, la velocidad y los kilómetros recorridos.

4.4 Conflictos encontrados

Al principio nos costó entender la jerarquía de control con este patrón ya que todas las clases siguen una estructura de torre, dependiendo cada una de la de encima. Sin embargo, una vez comprendida no tuvimos mayor problema para implementarlo salvo en el caso de los paneles que se muestran, ya que en el guion se habla de jPanel pero ya que no conseguíamos que se mostraran como queríamos los hemos cambiado por jFrames, lo cual no modifica la funcionalidad del código y representa con mayor claridad la interfaz.

5. Aplicación de WebScraping.

5.1 Introducción

Al tener poca práctica con Python y el termino WebScraping, hicimos una investigación en distintos foros como Medium y VerneAcademy acerca de **BeautifulSoup** y **Selenium**. Con esta información pudimos ir desarrollando gradualmente el problema.

La principal diferencia entre ambas es que Beautiful Soup es ideal para extraer datos estáticos de páginas web, mientras que Selenium es más adecuado para interactuar con páginas web dinámicas y automatizar acciones complejas dentro del navegador.

5.2 Solución del problema

5.2.1 BeautifulSoup

En el código hemos añadido comentarios adicionales; aquí explicaremos las funciones más relevantes.

En primer lugar, importamos los módulos necesarios y definimos una clase principal llamada **WebScraper**, que tendrá un constructor `__init__` que tomara un parámetro estrategia que será la que elijamos entre Selenium y BeautifulSoup. Además de la clase *extraer_datos(self, símbolo)* que extrará la información.

Comenzamos con **BeautifulSoup** que en un principio era la más fácil de desarrollar, básicamente construimos una url a la que nos queremos conectar, extraemos y analizamos la información de esa página y la guardamos en una variable **soup** en formato texto. En este punto nos surgió el problema de cómo extraer la información concreta de la página y después de buscar encontramos que se podía ver el nombre de las variables con el método inspeccionar del navegador.

Con esta información tenemos:

```
cierre_previo = soup.find("td", {"data-test": "PREV_CLOSE-value"}).text
```

El método **soup.find** busca en *soup* un elemento < td > (Celda de una tabla) que tenga un atributo específico *data-test* con un valor determinado "PREV_CLOSE-value". Repetimos esto con las demás variables.

5.2.2 Selenium

En principio **Selenium** es muy parecido a BeautifulSoup pero necesita una configuración más compleja a la hora de importar módulos y desarrollo, en el portal Selenium with Python pudimos encontramos los imports necesarios de modo que tenemos:

```
from selenium.webdriver.chrome.options import Options #Configurar el navegador rn Selenium
from selenium.webdriver.common.by import By #Enumeración que define los mecanismos de localización de elementos en Selenium
from selenium.webdriver.support.ui import WebDriverWait #Esperas explicitas
from selenium.webdriver.support import expected_conditions as EC #Expected Condition
```

Al igual que en BeautifulSoup definimos un *extraer_datos* en la que creamos una url pero tenemos que configurar Selenium. Mediante *options* establecemos **headless** en true para no tener interfaz gráfica y ocultar el navegador.

```
# Configuración de Selenium
        options = Options()
```

```
        options.headless = True
        driver = webdriver.Chrome(options=options)
```

Por último extraemos la información, lo dividimos en 2 partes de las cuales la segunda fue la más dificil, gracias a un foro de StackOverflow pudimos solucionarlo:

```
    1. wait = WebDriverWait(driver, 10): Con esto nos aseguramos de que los elementos HTML que estamos buscando estén presentes en la página antes de intentar extraer su contenido

    2. cierre_previo = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, "td[data-test='PREV_CLOSE-value']"))).text: localizamos un elemento utilizando un selector CSS mediante una condición de esperada que indica que Selenium debe esperar hasta que el elemento td sea visible en la página antes de continuar.
```

5.2.3 Main

```
1. simbolo = "TSLA": Creamos un simbolo con el nombre de la accion
2. web_scraper = WebScraper(Estrategia): Estrategia usada
3. dataX = web_scraper.extraer_datos(simbolo): Scraping y obtención de datos
4. json.dump(dataX, open("salidaX.json", "w")): Guardado de los datos en un archivo JSON
```

Al ejecutar en algunas ocasiones hay que darle al botón de X para detener el navegador y continuar con la ejecución.

5.3 Conclusión

Tras este ejercicio podemos determinar que BeautifulSoup no ejecuta JavaScript y solo accede al contenido HTML inicial mientras que Selenium si lo hace, accediendo al contenido dinámico y a elementos interactivos.

En términos de velocidad BeautifulSoup es más rápido al no cargar un navegador completo y Selenium puede ser más lento, especialmente en modo visible, al cargar el navegador y simular interacciones.

Se ha usado el patrón Strategy para permitir al programa elegir entre dos formas de realizar el web scraping. Este patrón separa las estrategias de scraping de la clase principal, lo que hace que el código sea más flexible. Así, se pueden agregar nuevas estrategias fácilmente y cambiar entre ellas sin alterar el código principal.